

# Komunikacja szeregową i symulacja komunikacji

Bogdan Kreczmer

bogdan.kreczmer@pwr.wroc.pl

Katedra Cybernetyki i Robotyki  
Wydziału Elektroniki, Fotoniki i Mikrosystemów  
Politechnika Wrocławska

*Kurs: Wizualizacja danych sensorycznych*

Copyright©2022 Bogdan Kreczmer

---

*Niniejszy dokument zawiera materiały do wykładu dotyczącego programowania obiektowego. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych prywatnych potrzeb i może on być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.*

*Niniejsza prezentacja została wykonana przy użyciu systemu składu  $\text{\LaTeX}$  oraz stylu beamer, którego autorem jest Till Tantau.*

Strona domowa projektu Beamer:

`http://latex-beamer.sourceforge.net`

# termios

**termios** – definiuje API Unix dla terminalowych urządzeń I/O. został on wprowadzony w ramach standardu POSIX.

Przed wprowadzeniem POSIX parametry transmisji do urządzeń terminalowych były obsługiwane poprzez funkcję **ioctl()**.

# Struktura termios

```
struct termios {  
    tcflag_t c_iflag;    /* input specific flags (bitmask) */  
    tcflag_t c_oflag;    /* output specific flags (bitmask) */  
    tcflag_t c_cflag;    /* control flags (bitmask) */  
    tcflag_t c_lflag;    /* local flags (bitmask) */  
    cc_t      c_cc[NCCS]; /* special characters */  
};
```

# Znaki specjalne

```
termios_set.c_cc[VINTR]      = 0;   /* Ctrl-c */
termios_set.c_cc[VQUIT]     = 0;   /* Ctrl-\ */
termios_set.c_cc[VERASE]    = 0;   /* del */
termios_set.c_cc[VKILL]     = 0;   /* @ */
termios_set.c_cc[VEOF]      = 0;   /* Ctrl-d */
termios_set.c_cc[VTIME]     = 0;   /* będzie używany wewnętrzny stoper */
termios_set.c_cc[VMIN]      = 1;   /* czyta gdy jest co najmniej 1 znak */
termios_set.c_cc[VSWTC]     = 0;   /* '\0' */
termios_set.c_cc[VSTART]    = 0;   /* Ctrl-q */
termios_set.c_cc[VSTOP]     = 0;   /* Ctrl-s */
termios_set.c_cc[VSUSP]     = 0;   /* Ctrl-z */
termios_set.c_cc[VEOL]      = 0;   /* '\0' */
termios_set.c_cc[VREPRINT]  = 0;   /* Ctrl-r */
termios_set.c_cc[VDISCARD]  = 0;   /* Ctrl-u */
termios_set.c_cc[VWERASE]   = 0;   /* Ctrl-w */
termios_set.c_cc[VLNEXT]    = 0;   /* Ctrl-v */
termios_set.c_cc[VEOL2]     = 0;   /* '\0' */
```

# Otwieranie portu komunikacji szeregowej

```
...  
int DeskPortu;  
  
if ((DeskPortu = open("/dev/ttyUSB0", O_RDWR | O_NONBLOCK)) < 0) {  
    cerr << ":( _Bład _otwarcia _portu _USB0" << endl;  
    return false;  
}  
...
```

# Ustawianie parametrów

```
bool SetTransParam( int          PortDesc ,
                    speed_t      BRate = B115200 ,
                    Type4Parity  Parity = P_None ,
                    int          CSize = CS8
                    )
{
    struct termios  TransParam;

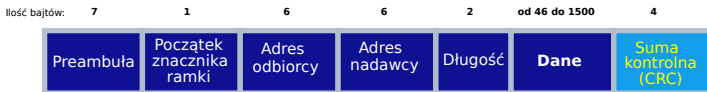
    TransParam.c_iflag = IGNBRK | (Parity != P_None ? INPCK : 0);
    TransParam.c_oflag = NL0 | CR0 | FF0;
    TransParam.c_cflag = CSize | CREAD
        | (Parity != P_None ? PARENB : 0)
        | (Parity == P_Odd ? PARODD : 0) | CLOCAL
        | (Parity == P_Mark ? CMSPAR : 0 );
    TransParam.c_lflag = NOFLSH;
    for (int i=0; i < NCCS; ++i) TransParam.c_cc[i] = 0;
    cfsetispeed(&TransParam, BRate);
    cfsetospeed(&TransParam, BRate);
    tcflush(PortDesc, TCIFLUSH);
    return tcsetattr(PortDesc, TCSANOW, &TransParam) == 0;
}
```

# Kontrola spójności danych

## CRC (Cyclic Redundancy Check)

Blok danych można zabezpieczyć dodatkową wartością, która pozwala wykrycie ewentualnych błędów transmisji. Należy pamiętać, że tego typu zabezpieczenie jest odwzorowanie *na*. Oznacza to, że różnym blokom danych może odpowiadać ta sama wartość.

W najprostszym ujęciu wartość CRC jest resztą z dzielenia przez pewien wielomian. Wielkość i rodzaj wielomianu zależą od wielkości bloku danych i spodziewanych błędów, które mają być wykryte.



Ramka danych dla Ethernet: IEEE 802.3



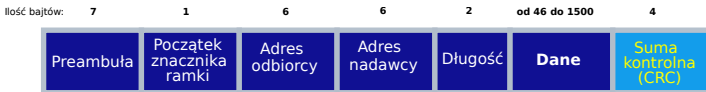
# Kontrola spójności danych – trochę teorii

## CRC (Cyclic Redundancy Check)

Metoda wyliczania CRC jest oparta na teorii cyklicznych kodów korygujących błędy. Idea dodawania wartości kontrolnej o stałej długości została zaproponowana przez W. Wesley Peterson w 1961.

Operacje dzielenia wielomianów realizuje się w oparciu o teorię ciał skończonego rzędu (ciało Galois).

*Sumę kontrolną* w tym podejściu definiuje się jako  $n$ -bitowy cykliczny kod nadmiarowy ( $n$ -bitowy CRC), która jest resztą z dzielenia ciągu danych przez  $(n+1)$ -bitowy dzielnik CRC nazywany wielomianem CRC.



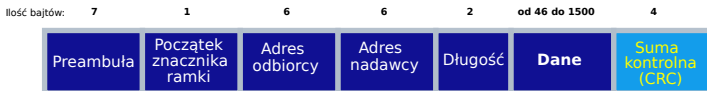
Ramka danych dla Ethernet: IEEE 802.3

# Kontrola spójności danych

## CRC-N

Wielkość najczęściej stosowanych wielomianów:

- 9-bitowy → CRC-8,
- 17-bitowy → CRC-16,
- 33-bitowy → CRC-32,
- 65-bitowy → CRC-64,



Ramka danych dla Ethernet: IEEE 802.3

# Dzielenie (*w arytmetyce XOR*) – krok po kroku

-----

111010 : 110

# Dzielenie (w arytmetyce *XOR*) – krok po kroku

```
  1
-----
```

```
111010 : 110
```

```
110
```

<- Na pozycji odpowiadającej przesunięciu wpisujemy 1

<- Dzielnik przesuwamy do lewej skrajnej pozycji

# Dzielenie (w arytmetyce *XOR*) – krok po kroku

```
      1
-----
111010 : 110
110000      <- Uzupełniamy zerami i dokonujemy operacji XOR
  1010      <- Wynik operacji
```

# Dzielenie (w arytmetyce *XOR*) – krok po kroku

Na pozycji odpowiadającej przesunięciu wpisujemy 1,  
<- zaś w pozycjach poprzedzających wpisujemy 0.

```
    101
-----
111010 : 110
110000
  1010
   110
```

<- Powtarzamy operację przesunięcia

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
101
-----
111010 : 110
110000
 1010
 1100      <- Uzupełniamy zerami i dokonujemy operacji XOR
 110       <- Wynik operacji
```

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
101
-----
111010 : 110
110000
 1010
 1100      <- Uzupełniamy zerami i dokonujemy operacji XOR
 110       <- Wynik operacji
```



# Dzielenie (w arytmetyce XOR) – krok po kroku

Na pozycji odpowiadającej pozycji dzielnika  
<- wpisujemy 1.

1011  
-----

111010 : 110

110000

1010

1100

110

110

<- Przepisujemy dzielnik, tym razem nie trzeba  
go przesunąć.

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
    1011
-----
111010 : 110
110000
  1010
   1100
    110
     110
      0
```

<- Wynik 0 kończy całą operację.

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
  1011
*  110
-----
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
    *   110
    -----
    1011      <- 1011*100 (2^3+2^1+2^0)*2^2
     1011     <- 1011*100 (2^3+2^1+2^0)*2^1
    =====
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
    *   110
    -----
      1011
     1011
    =====
      1
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
    *   110
    -----
      1011
      1011
    =====
       11
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
*     110
-----
      1011
      1011
=====
      111
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
*     110
-----
      1011
      1011
=====
      1110
```



# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
    *   110
    -----
      1011
      1011
    =====
      11101
```

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

111010 : 110 = 1011

```
      1011
*     110
-----
      1011
      1011
=====
      111010
```

<- Końcowy wynik

# Dzielenie z resztą (*w arytmetyce XOR*)

-----

111010+1 : 110

# Dzielenie z resztą (*w arytmetyce XOR*)

-----

111011 : 110

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
      1011
-----
111011 : 110
110000
  1011
   1100
    111
     110
      1      <- Reszta
```

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
1011
-----
11101100 : 110  <- Resztę wyznaczamy dla liczby rozszerzonej o 2 zera
110000
 1011
 1100
  111
  110
   100
```

Ogólnie ilość dopisywanych zer odpowiada stopniowi wielomianu będącego dzielnikiem. W tym przypadku jest to stopień drugi, tzn.

$$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

# Dzielenie (w arytmetyce XOR) – krok po kroku

```
      1011
-----
11101100 : 110
110000
  1011
   1100
    111
     110
      100
       110
```

<- Przepisujemy dzielnik i przesuwamy o DWIE pozycje w prawo i wykonujemy operację XOR

# Dzielenie z resztą (w arytmetyce XOR)

```
101100
-----
11101100 : 110
110000
1011
1100
111
110
100
110
10      <- To jest reszta z dzielenia
```



# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

11101100 : 110 = 101101 (reszta: 10)

101101  
(xor) \* 110  
-----

101101  
(xor) + 101101  
=====

11101110

101101\*100 {  $(2^5+2^3+2^2+2^0)*2^3$  }

101101\*10 {  $(2^5+2^3+2^2+2^0)*2^2$  }

<- Wynik bez uwzględnienia reszty

# Mnożenie (w sensie arytmetyki XOR) – sprawdzenie

11101100 : 110 = 101101 (reszta: 10)

101101  
(xor) \* 110  
-----

101101            101101\*100 { (2<sup>5</sup>+2<sup>3</sup>+2<sup>2</sup>+2<sup>0</sup>)\*2<sup>3</sup> }  
(xor) + 101101    101101\*10 { (2<sup>5</sup>+2<sup>3</sup>+2<sup>2</sup>+2<sup>0</sup>)\*2<sup>2</sup> }  
=====

11101110        <- Wynik bez uwzględnienia reszty  
(xor) + 10        <- Dodajemy resztę

=====

11101100        <- Końcowy wynik

# Przykładowa implementacja CRC8

## Ogólna idea

W dalszej części przedstawione są dwie główne funkcje, które implementują obliczanie CRC8. Są to: `CRC8_SingleByte(.)` i `CRC8_DataArray`.

Implementują one niemal wprost procedurę dzielenia z tą różnicą, że w procesie dzielenia **przesuwana jest dzielna, a nie dzielnik**.

Funkcja `CRC8_SingleByte(.)` realizuje operację dzielenia starszego bajtu dwubajtowego słowa. W efekcie tej operacji młodszy bajt zostaje przesunięty w miejsce starszego.

Funkcja `CRC8_DataArray` powoduje użycie funkcji `CRC8_SingleByte(.)` sukcesywnie do każdego bajtu danej sekwencji bajtów.

Przedstawiony przykład jest najprostszy, ale też najmniej efektywny. Niemniej w zupełności wystarczy, gdy zależności czasowe nie są bardzo krytyczne.

# Przykładowa implementacja CRC8

```
typedef unsigned char byte;  
...  
  
byte CRC8_SingleByte(byte CRC_prev, byte Data)  
{  
    ...  
}
```

```
byte CRC8_DataArray(byte *pData, byte Len)  
{  
    unsigned int Data2 = pData[0] << 8;  
  
    for (unsigned int Idx = 1; Idx < Len; ++Idx) {  
        Data2 |= pData[Idx];  
        Data2 = CRC8_SingleByte(Data2);  
    }  
    Data2 = CRC8_SingleByte(Data2);  
  
    return static_cast<byte>(Data2 >> 8);  
}
```

# Przykładowa implementacja CRC8

```
typedef unsigned char byte;  
#define POLYNOMIAL_9 0x0161  
  
unsigned int CRC8_SingleByte(unsigned int Data2)  
{  
    unsigned int Poly = (POLYNOMIAL_9 << 8);  
  
    for ( byte ldx = 0; ldx < 8; ++ldx ) {  
        Data2 <<= 1;  
        if ( ( Data2 & 0x10000 ) != 0 ) Data2 ^= Poly;  
    }  
    return Data2;  
}
```

```
byte CRC8_DataArray(byte *pData, byte Len)  
{  
    ...  
}
```

Tylko wtedy gdy sizeof(int) > 2

# Przykładowa implementacja CRC8

```
typedef unsigned char byte;  
#define POLYNOMIAL_9 0x0161  
  
unsigned int CRC8_SingleByte(unsigned int Data2)  
{  
    unsigned int Poly = (POLYNOMIAL_9 << 7);  
  
    for ( byte ldx = 0; ldx < 8; ++ldx) {  
        if ( ( Data2 & 0x8000 ) != 0 ) Data2 ^= Poly;  
        Data2 <<= 1;  
    }  
    return Data2;  
}
```

```
byte CRC8_DataArray(byte *pData, byte Len)  
{  
    ...  
}
```

Tylko wtedy gdy sizeof(int) >= 2  
np. dla 16-bitowego mikrokontrolera.

# Jak to działa – krok po kroku

Dla uproszczenia przykładu przyjmijmy, że mamy skrócone bajty do 4 bitów.

Zamiast CRC-8 będziemy liczyć CRC-4    Wielomian CRC: 11011  
Przesyłana informacja dla której  
trzeba obliczyć CRC to:        1010 0100 1001 0110

Traktujemy całość jako jedną liczbę i dopisujemy 4 zera (ilość zer równy jest stopniowi wielomianu).  
Trzeba więc obliczyć resztę z dzielenia:

1010 0100 1001 0110 0000    : 11011

Działanie funkcji CRC8\_SingleByte zostanie wyjaśnione na przykładzie analogicznej funkcji CRC4\_SingleByte().

```
#define POLYNOMIAL_5 0x01B

unsigned int CRC4_SingleByte(unsigned int Data2)
{
    unsigned int Poly = (POLYNOMIAL_5 << 3);

    for ( byte idx = 0; idx < 4; ++idx ) {
        if ( ( Data2 & 0x80 ) != 0 ) Data2 ^= Poly;
        Data2 <<= 1;
    }
    return Data2;
}
```

Działanie tej funkcji realizuje procedurę dzielenia z tą różnicą, że przesuwą dzielną zamiast dzielnik w tradycyjnym pisemnym dzieleniu.

# Jak to działa – krok po kroku

```
unsigned int CRC4_SingleByte(unsigned int Data2)
{
    unsigned int Poly = (POLYNOMIAL_5 << 3);

    for ( byte Idx = 0; Idx < 4; ++Idx ) {
        if ( ( Data2 & 0x80 ) != 0 ) Data2 ^= Poly;
        Data2 <<= 1;
    }
    return Data2;
}
```

Przed pierwszym wywołaniem do parametru Data2 wpisywane są dwa pierwsze "bajty" transmitowanego ciągu, czyli: 1010 0100

Działanie funkcji	<----->	Dzielenie XOR
Poniżej kolejne iteracje pętli for		W miejsce kropek są zera. Dla przejrzystości zapisu oznaczono
Kolejne iteracje:		je kropkami zamiast w ogóle pominąć, jak w pisemnym dzieleniu.
=====		
1. Data2: 1010 0100	+-----+	1010 0100 1001 0110 0000 : 11011
Poly: 1101 1000		1101 1000 .... .... <- dzielnik
-----		
Najstarszy bit Data2 jest 1, więc wykonujemy XOR		
-----		
Data2: 0111 1100 : <- wynik XOR		111 1100 1001 0110 0000
Data2: 1111 1000 : <- Data2 <<=1		
=====		
2. Data2: 1111 1000		
Poly: 1101 1000		110 11.. .... .... <- dzielnik
-----		
Najstarszy bit Data2 jest 1, więc wykonujemy XOR		
-----		



# Jak to działa – krok po kroku

```
unsigned int CRC4_SingleByte(unsigned int Data2)
{
    unsigned int Poly = (POLYNOMIAL_5 << 3);

    for ( byte Idx = 0; Idx < 4; ++Idx ) {
        if ( ( Data2 & 0x80 ) != 0 ) Data2 ^= Poly;
        Data2 <<= 1;
    }
    return Data2;
}
```

Działanie funkcji

&lt;-----&gt;

Dzielenie XOR

...	...
=====	
2. Data2: 1111 1000	111 1100 1001 0110 0000
Poly: 1101 1000	
-----	110 11.. .... .... <- dzielnik
Najstarszy bit Data2 jest 1, więc wykonujemy XOR	
-----	-----
Data2: 0010 0000 : <- wynik XOR	1 0000 1001 0110 0000
Data2: 0100 0000 : <- Data2 <<=1	
=====	
3. Data2: 0100 0000	
-----	
Najstarszy bit Data2 jest 0, więc pomijamy XOR	
-----	
Data2: 1000 0000 : <- Data2 <<=1	
=====	
4. Data2: 1000 0000	
Poly: 1101 1000	1 1011 000. .... ....
-----	-----

Wizualizacja danych sensorycznych

Komunikacja szeregową ...

# Jak to działa – krok po kroku

```
unsigned int CRC4_SingleByte(unsigned int Data2)
{
    unsigned int Poly = (POLYNOMIAL_5 << 3);

    for ( byte Idx = 0; Idx < 4; ++Idx ) {
        if ( ( Data2 & 0x80 ) != 0 ) Data2 ^= Poly;
        Data2 <<= 1;
    }
    return Data2;
}
```

Działanie funkcji

&lt;-----&gt;

Dzielenie XOR

```
...
=====
4. Data2: 1000 0000
   Poly: 1101 1000
-----
Najstarszy bit Data2 jest 0, więc pomijamy XOR
-----
   Data2: 0101 1000
   Data2: 1011 0000 : <- Data2 <<=1
=====
```

W tym miejscu kończy się działanie pętli for i ostatnia wartość Data2 jest zwracana.  
Następnie do młodszego bajtu wpisywany jest kolejny bajt komunikatu i funkcja jest ponownie wywoływana tym razem dla wartości: 1011 1001

Po wykonaniu tej sekwencji kroków dla całego komunikatu otrzymujemy resztę równą: 1110

# CRC – ciekawe artykuły

## Cykl artykułów

Autor: Jarosław Doliński

- *CRC doda Ci pewności, część 1*, Elektronika Praktyczna, 1/2003  
<https://ep.com.pl/files/4641.pdf>
- *CRC doda Ci pewności, część 2*, Elektronika Praktyczna, 2/2003  
<https://ep.com.pl/files/4656.pdf>
- *CRC doda Ci pewności, część 3*, Elektronika Praktyczna, 3/2003  
<https://ep.com.pl/files/4681.pdf>
- *CRC doda Ci pewności, część 4*, Elektronika Praktyczna, 4/2003  
<https://ep.com.pl/files/4724.pdf>
- *CRC doda Ci pewności, część 5*, Elektronika Praktyczna, 5/2003  
<https://ep.com.pl/files/4730.pdf>

Artykuły zawierają opis szybkich algorytmów opartych na tablicach, jak też znaczenia ogólnych parametrów dla tego typu metod.

# Przykład definicji ramki danych

## Założenia

- Transmisja jest znakowa.
- Nagłówkiem ramki jest znak 'X'.
- Ramka nie ma określonej ilości znaków.
- Terminatorem ramki jest sekwencja <CR><LF>.
- Wartości liczbowe zapisywane są w systemie dziesiętnym.
- Separatorem jest znak spacji.
- Ostatnią wartością jest CRC8 i zapisywana jest w notacji heksadecymalnej.

# Przykład parsowania ramki

Przykład zawartości ramki bez <CR><LF>

X 147 45281 FC

```
bool ParseDataFrame(const char *pDataFrame, int &AccX, int &AccY)
{
    std::istringstream IStrm(pDataFrame);
    char               FHeader;
    unsigned int        CRC8;

    IStrm >> FHeader >> AccX >> AccY >> hex >> CRC8;
    if (IStrm.fail() || FHeader != 'X') return false;
    return static_cast<byte>(CRC8) ==
           CRC8_DataArray(reinterpret_cast<byte*>(pDataFrame),
                        strlen(pDataFrame)-3
                        );
}
```

# Najważniejsze cechy

Pseudoterminal (skrót *pseudotty* lub *PTY*) jest parą urządzeń wirtualnych. Jedno z nich pełni rolę urządzenia podrzędnego (*slave*), drugie zaś jest urządzeniem nadrzędnym *master*.

*Slave* emuluje zwykły terminal znakowy. Pozwala więc odczytać wprowadzone do niego teksty i reaguje na sekwencje sterujące. Umożliwia też wprowadzanie tekstów.

*Master* dostarcza narzędzi, które pozwalają kontrolować *slave* przez proces emulatora terminala.

Zadania procesu emulatora terminala:

- interakcja z użytkownikiem,
- wprowadzenie tekstu do *mastera* i dostarczenie go powłocie (np. *bash*) poprzez *slave'a*, z którym połączona jest powłoka,
- czytanie tekstu z wyjścia *mastera* i pokazanie go użytkownikowi.

# Interfejsy programistyczne

## Przykłady użycia pseudoterminala

- Sesja z terminalem tekstowym
- Otwarcie okienka graficznego z terminalem w środowisku graficznym
- Zdalna sesja tekstowa poprzez ssh

Dla pseudoterminali ukształtowały się dwa główne interfejsy programistyczne. Wywodzą się one z systemów:

- BSD (Berkeley Software Distribution) – odmiana Uniksa bazująca na stworzonych w Kalifornijskim Uniwersytecie Berkeley rozszerzeniach wersji systemu opracowanego przez firmę AT&T.
- System V – wersja systemu UNIX rozwijana w Bell Labs firmy AT&T. Nazwę *System V* używa się od wersji Uniksa wydawanych przez AT&T od 1983.

# Pseudoterminale w Linuksie

Systemy nazewnictwa i interfejsy w systemach Uniksowych są znormalizowane normami takimi jak UNIX 95, UNIX 98, UNIX 03, UNIX V7. Nowsze normy oparte są na normach POSIX.

Systemy uniksopodobne, takie jak Linux nie spełniają w pełni żadnej z tych norm. Domyślnie też nie są instalowane wszystkie narzędzia zdefiniowane przez normę POSIX.

W systemach linuksowych wykorzystywany jest interfejs pseudotermini zdefiniowany w ramach normy UNIX 98 oraz późniejszych. Te normy nie ograniczają ilość pseudotermini.



# Tworzenie pseudoterminala na poziomie powłoki

## Tworzenie powiązanych pseudoterminali

Przykład tworzenia pseudoterminala na poziomie powłoki. Do utworzenia powiązanych ze sobą dwóch pseudoterminali można wykorzystać program socat.

```
jkowalsk@linux> socat -d -d pty,raw,echo=0 pty,raw,echo=0
2018/05/03 16:21:49 socat[17186] N PTY is /dev/pts/5
2018/05/03 16:21:49 socat[17186] N PTY is /dev/pts/6
2018/05/03 16:21:49 socat[17186] N starting data transfer loop
with FDs [5,5] and [7,7]
```

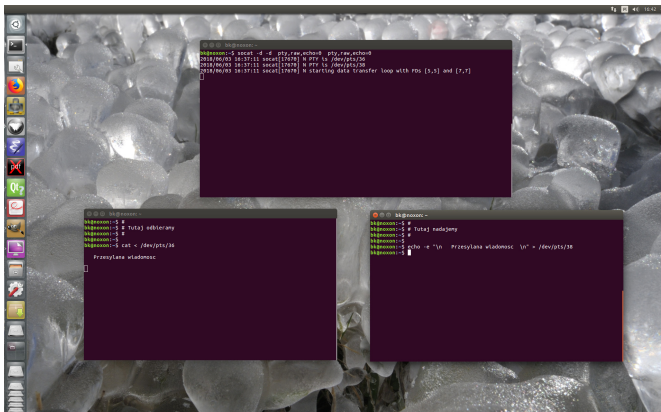
W okienkach terminali tekstowych możemy przetestować komunikację w następujący sposób:

```
cat < /dev/pts/5
```

```
echo "Test" > /dev/pts/6
```

# Tworzenie pseudoterminala na poziomie powłoki

Wzajemną komunikację między pseudoterminalami można przetestować na poziomie powłoki, tak jak to jest pokazane poniżej. Można też odwoływać się do nich na poziomie programu, tak jak do normalnego portu szeregowego. W tym celu można użyć poleceń z pakietu `termios` lub klasy `QSerialPort`.



```
bb@bbcon-:~$ socat -d -d pty,raw,echo=0 pty,raw,echo=0
2018/06/03 14:37:11 socat[17078] N pty ts /dev/pts/36
2018/06/03 14:37:11 socat[17078] N pty ts /dev/pts/36
2018/06/03 14:37:11 socat[17078] N starting data transfer loop with PDS [5,5] and [7,7]

bb@bbcon-:~$
bb@bbcon-:~$ #
bb@bbcon-:~$ # Tutaj odbieramy
bb@bbcon-:~$ #
bb@bbcon-:~$ cat < /dev/pts/36
Przesłana wiadomość

bb@bbcon-:~$
bb@bbcon-:~$ # Tutaj nadajemy
bb@bbcon-:~$ #
bb@bbcon-:~$ echo -e "\n Przesłana wiadomość \n" > /dev/pts/36
bb@bbcon-:~$
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

Kluczowym urządzeniem wirtualnym jest *pseudo-terminal master multiplexer* (`/dev/ptmx`).  
Utworzenie pseudoterminala składa się z następujących kroków:

- Wykonana zostaje operacja otwarcia urządzenia wirtualnego `/dev/ptmx`. Powoduje to zwrócenie deskryptora pliku do węzła *master*.  
`O_RDWR` – otwarcie w trybie czytania i zapisu,  
`O_NOCTTY` – nie będzie procesem sterującym terminalem.

```
int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);  
if (fd == -1) {  
    std::cerr << "error opening file" << std::endl;  
    return -1;  
}
```

```
grantpt(fd);  
unlockpt(fd);
```

```
char* pts_name = ptsname(fd);  
std::cout << "ptsname: " << pts_name << std::endl;
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

- Zostaje utworzone wirtualne urządzenie *slave* `/dev/pts/N`, które jest skojarzone z utworzonym węzłem *master*.

```
int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
if (fd == -1) {
    std::cerr << "error_opening_file" << std::endl;
    return -1;
}

grantpt(fd);
unlockpt(fd);

char* pts_name = ptsname(fd);
std::cout << "ptsname: " << pts_name << std::endl;
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

- Funkcja `grantpt` zmienia tryb dostępu i właściciela urządzenia *slave* będącego składnikiem pseudoterminala. Właścicielem staje się użytkownik, do którego należy dany proces. Tryb dostępu zostaje ustawiony na `crw--w----`.

```
int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
if (fd == -1) {
    std::cerr << "error opening file" << std::endl;
    return -1;
}

grantpt(fd);
unlockpt(fd);

char* pts_name = ptsname(fd);
std::cout << "ptsname: " << pts_name << std::endl;
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

- Funkcja `unlockpt` odblokowuje urządzenie *slave* pseudoterminala odpowiadającego deskryptorowi `fd`.

```
int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
if (fd == -1) {
    std::cerr << "error_opening_file" << std::endl;
    return -1;
}

grantpt(fd);
unlockpt(fd);

char* pts_name = ptsname(fd);
std::cout << "ptsname: " << pts_name << std::endl;
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

- Funkcja `ptsname` zwraca nazwę urządzenia *slave* pseudoterminala odpowiadającego deskryptorowi `fd`.

```
int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
if (fd == -1) {
    std::cerr << "Error_opening_file" << std::endl;
    return -1;
}

grantpt(fd);
unlockpt(fd);

char* pts_name = ptsname(fd);
std::cout << "ptsname: " << pts_name << std::endl;
```

# Tworzenie pseudoterminala

## Przykłady użycia pseudoterminala

- W drugim programie można już otworzyć w sposób standardowy komunikację z aplikacją poprzez podane urządzenie. W tym przykładzie było to: `/dev/pts/20`

```
int fd = open("/dev/pts/20", O_RDWR | O_NOCTTY);
if (fd == -1) {
    std::cerr << "Error opening file" << std::endl;
    return -1;
}
const char *sNapis = "_Czesc_swiecie_\r\n";
write(fd, sNapis, strlen(sNapis));
```



Koniec prezentacji  
Dziękuję za uwagę